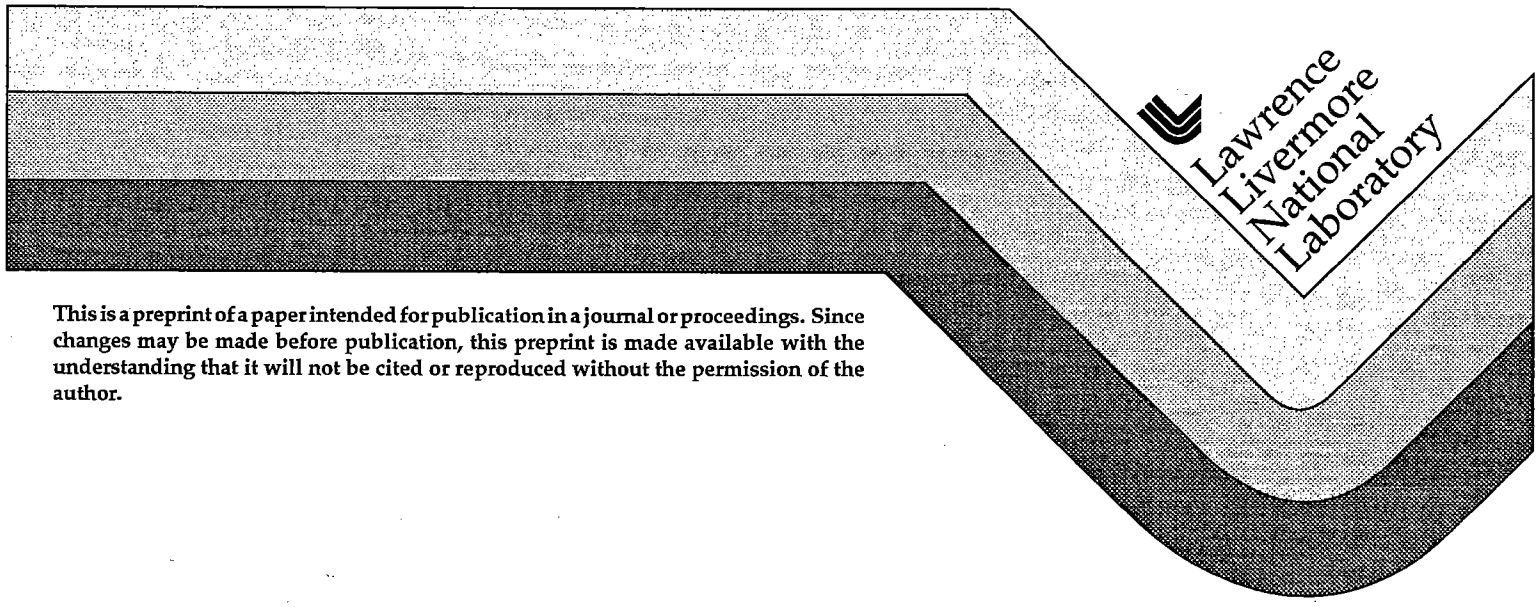
UCRL-JC-115647
PREPRINT

Hybrid Ordered Particle Simulation (HOPS) Code for Plasma Modeling on Vector-Serial, Vector-Parallel, and Massively Parallel Computers

D.V. Anderson
D.E. Shumaker

This paper was prepared for submittal to
Computer Physics Communications

November 1993



DISCLAIMER

HYBRID ORDERED PARTICLE SIMULATION (HOPS) CODE FOR PLASMA MODELLING ON VECTOR-SERIAL, VECTOR-PARALLEL, AND MASSIVELY PARALLEL COMPUTERS

David V. Anderson and Dan E. Shumaker

National Energy Research Supercomputer Center
Lawrence Livermore National Laboratory
Livermore, California 94550 USA

ABSTRACT

Many computational models of plasmas, and other physical systems, have employed the particle-in-cell (PIC) method in which simulation particles are followed in the self-consistent physical fields represented on a grid. In the case of plasmas such models have allowed detailed investigations of kinetic phenomena which are difficult or impossible to model with a fluid treatment. These particle codes conventionally store the simulation particle information in tables in which the particle spatial positions tend to be random with respect to the position in the table. With the advent of vector computers the relative efficiencies of particle codes versus fluid codes began to suffer and now with the development of parallel computers these kinds of codes are in need of radical redesign to enable them to run efficiently. Problems associated with excessive accessing of memory, indirect indexing, and with many-to-one mappings in the deposition phase can make these codes inefficient on vector-serial, vector-parallel, and massively parallel machines. In the HOPS (Hybrid Ordered Particle Simulation) code we employ a sorting scheme to keep the particles ordered with respect to their spatial positions. By doing so, we have reduced memory accesses, recovered substantial direct indexing, and most importantly removed the many-to-one mapping problem. Even with the overhead of sorting and reordering the particle tables, we have developed a code which is most efficient on vector-serial machines and which linearly scales to various kinds of parallel computers. We verify the operation of HOPS by presenting example problems involving ionospheric plasmas and ones of interest in magnetic fusion research. This paper focuses on the Cray C-90 vector-parallel computer but also presents information on massively parallel implementations.

1 Introduction

Perhaps the simplest and most general methodology for simulating the behavior of plasmas is to follow the charged particles in their self-consistent electromagnetic fields.[1,2,3] To do this, one solves the coupled Newton-Maxwell equations for the modeled problem. Though straightforward, this

approach is very intensive computationally both in regard to storage and time requirements. Thus, historically, most models of plasmas have employed a fluid representation inorder to make the calculations affordable. These particle simulation models are employed in the studies of astrophysics, particle accelerators, solid state physics, molecular dynamics, as well as plasma physics. In this paper we introduce the HOPS code which has been designed to operate efficiently in contemporary scientific computing environments. Although the presentation is given with respect to a version running on the Cray C-90 vector-parallel machine we also indicate its relevance to massively parallel computers.

A typical particle code is executed in four phases:

1. Interpolate the electromagnetic fields from the grid to the particle positions (*field interpolation*)
2. Using these fields, push particles by advancing positions and velocities according to Newton's law (*push*)
3. Interpolate from particles to deposit charge and current densities on the grid (*deposition interpolation*)
4. Solve the Maxwell field equations on the grid (*solve*)

Most particle codes are actually hybrid codes in which some of the species are treated as fluids. The codes we study here are all of this hybrid type. Except for the special fluid model used for the electrons, we shall not describe the treatment of the other fluid components except to note that we use the FCT method pioneered by Boris[5] as developed by Zalesak.[6]

It is true for many particle codes, and it will be true here as well, that the phase of *deposition interpolation* will severely constrain our strategy for optimization of HOPS. By optimization we mean to include such techniques as vectorization and parallelization in addition to the usual considerations. Issues arising from the first three phases have driven the choice of the scheme we employ, but it turns out that our field *solve* optimizes very well in the domain decomposition that results.

Prior to 1985 most particle codes were only partially vectorized, partly because the machines available had limited vector hardware and partly because the algorithms were not well developed. A notable exception to this was the work of Buneman et al[7] who wrote highly optimized particle codes in Cal assembly language for the Cray-1 computer. This situation began to change with the introduction of vector multiprocessor machines such as the Cray X-MP, Cray-2, Cray Y-MP, as well as some of Japanese manufacture. In the late 1980's, nearly all of the restrictions on vectorization and parallelization (multitasking) were removed. The hybrid particle code QN3D, with fluid electrons and particle ions, pioneered several of these developments[8] and was fully vectorized and parallelized for execution on a four processor Cray-2.

The first phase of the time step, as indicated in the list above, requires the interpolation of the fields, known on the mesh, to the particle positions. Since the particles, stored in so-called particle tables, tend to become randomly distributed in space, it follows that the fields used to move the particles must be accessed randomly from the computer memory. In terms of the written algorithm this is expressed in terms of an indirect index structure, such as *EX(I(N),J(N),K(N))*. (Here *I(N),J(N),K(N)* represents a grid point neighboring the *Nth* particle in the table and *EX* is the x component of the electric field.) The *push* phase is either combined with the *field interpolation* or is executed separately. It is the third phase, the *deposition interpolation*, which has frustrated optimization the most. The problem lies in the fact that several particles can contribute to the charge and current densities at each grid point. This has the form of a many-to-one mapping. Any attempt to deposit the particles' contributions concurrently can lead to a conflict; this conflict prevents most attempts to vectorize or parallelize the *deposition interpolation.*

In the last phase of HOPS we solve an approximate form of Maxwell's field equations. The field solve algorithm we employ is iteratively implicit which really means it is explicit at each iteration and therefore easily vectorized and parallelized.

1.1 Organization of this Paper

Following this Introduction we proceed, in Section II, to review various strategies that have been proposed for improving the performance of PIC codes. In section III we describe the original HOPS algorithm, which goes by the name OSOP (ordered storage ordered processing), and then go on to describe the version designated as RSOP (random storage ordered processing.) The vector-parallel generalizations are given in IV and our pre-Maxwell field solver is presented in V. Test problems are treated in Section VI while results and conclusions are offered in Section VII.

2 Strategies for High Performance PIC Codes

In our earlier attempts to improve the performance of PIC codes we were primarily interested in improving their serial performance on traditional vector supercomputers. In fact, the Cray-2, the Cray Y-MP, and the Cray C-90 have been the machines on which our studies have been performed. As a secondary consideration we also knew that we'd eventually want to run our applications in parallel so we tried to develop our serial code so that it would be amenable to parallelization at a later date. And from a standpoint of cost-effectiveness we have followed the advice that it is almost always better to optimize the serial performance of an algorithm before parallelizing it.

Our first observation about traditional PIC codes was that the arithmetic units of the CPU were mostly idle in the Cray-2 environment; most of the

work going on was the random accessing of the common memory to bring in or farm out quantities associated with the grid. It was also observed that each grid quantity was accessed many times in each time step; in fact each was accessed as many times as there were particles in the adjoining grid cells. So there were two problems in accessing the memory:

1. The memory locations were effectively random.
2. Each memory location was accessed many times in a given computational time step.

In subsequent tests of PIC codes on the Cray Y-MP and on the Cray C-90 we have found that the extremely "agile" memories of these two machines result in only a small degradation of the performance from excessive accessing of memory. Since the behavior of the memories of massively parallel computers is not well known, it is not clear whether the accessing of memory will "pace" the calculations or not.

2.1 Vectorization Including Gather-Scatter

The indirect index structure, such as *EX(I(N),J(N),K(N))*, which frustrated vectorization on the earlier vector computers, is now indirectly vectorized on most of the contemporary vector computers by gather and scatter operations. The gather mechanism is used to load memory items into the CPU vector registers while the related scatter operation is used to store into the memory from the CPU vector registers. The old QN3D code employed the gather procedure to allow vectorization and used multitasking to carry out the *field interpolation* phase. We note that this form of indirect vectorization is considerably slower, on the Cray-2, than normal direct vectorization of constant stride arrays; at best the gather operation can run at one fourth the normal vector speed. Even on the Cray Y-MP and Cray C-90 computers there is a significant degradation in the vector speed from indirect vectorization. The speed of indirect vectorization is lower because random access of memory tends to be slowed by memory bank conflicts. Yet, in most cases, it is still faster than scalar coding.

During the next phase, the particle *push*, the orbits are all independent and thus trivially vectorized and parallelized.

The third phase, the *deposition interpolation*, requires a more complicated cumulative interpolation phase and requires both a gather and a scatter operation to perform the deposition of charge and current density contributions from the various particles. There is a possible recursion in this step because different particles may try to increment the same mesh location simultaneously; consequently this aspect prevented vectorization and multitasking. This is the problem of the many-to-one mapping. One remedy to this problem, devised by Horowitz for the QN3D code[9], was to break the deposition phase into several sub-phases each of which accumulated the densities from a corresponding subgroup of particles. By choosing

each subgroup such that no two particles occupied the same mesh cell it then became impossible for conflicts to occur. QN3D used both vectorization and parallelization within each sub-phase to optimize the *deposition interpolation.*

2.2 Vectorization Avoiding Gather-Scatter

Other workers have also been developing schemes to improve the efficiency of particle codes on vector-serial computers. For example, Heron and Adam[10] developed a deposition algorithm which is organized such that most of the loops employ direct vectorization, thereby reducing much of the overhead of the alternative gather-scatter formulation.

2.3 Enhancing the Memory Access

In our experience, the least efficient type of memory access is the fetching (or storing) of single words from (or to) random memory locations. And the most efficient form of memory access moves continguous or constant stride arrays (or vectors.) Of particular importance to our project is the fact that the time to access short vectors, say from 4 to 12 words in length, is only marginally greater than that for accessing one word. This suggests the notion, borne out by tests, that random accessing of memory can be made significantly more efficient if we can move vectors of information instead of single words.

Since the *field interpolation* and the *deposition interpolation* both access several quantities in each "random" grid location, we are encouraged to use interleaving of the field and density arrays. Thus in the *field interpolation* we access the 12 quantities: $E_x(i,j,k)$, $E_y(i,j,k)$, $E_z(i,j,k)$, $B_x(i,j,k)$, $B_y(i,j,k)$, $B_z(i,j,k)$, $E_x(i+1,j,k)$, $E_y(i+1,j,k)$, $E_z(i+1,j,k)$, $B_x(i+1,j,k)$, $B_y(i+1,j,k)$, $B_z(i+1,j,k)$ as a vector of length 12 which is stored with unit stride because the arrays are interleaved. This is accomplished in Fortran by including a leading index which describes which 6-vector component to use from the 6-vector **E**,**B**. Thus the Fortran variable EX is dimensioned EX(6,IM,JM,KM) where IM, JM, and KM are the grid dimensions. This means that $E_y(i,j,k) =$ EX(2,I,J,K) and that $B_x(i,j,k) =$ EX(4,I,J,K), for example. A similar interleaving of the charge and current density arrays is also done in which the four quantities at each grid point form a 4-vector or when taking two adjacent grid points in x form an 8-vector. Interleaving the storage of the field arrays was first done by Langdon and Lasinski[4] in the mid 1970's to take advantage of the memory characteristics of the CDC-7600 large core memory. Though contemporary supercomputers differ markedly in their memory architecture, compared to the CDC-7600, they also have better memory performance when several contiguous (or constant stride) words of data are accessed together. As we shall describe in some detail later, it is the practice of interleaving together with a scheme for reducing the number of memory accesses that will result

in a particle simulation technique much more efficient in its memory use than was previously the case.

Interleaving the field arrays, and also the density arrays, not only helps the efficiency of the memory access, but it lends itself to the efficiency of the code by allowing direct vectorization in many of the loops.

The order of computation plays a big role in the amount of memory access that is needed. Traditionally, particle codes process the particles in the order that they occur in the particle tables. Since the spatial particle location and therefore the nearby grid quantities are relatively random, the various grid qunatities must be accessed every time we process another particle. If we could process all the particles within a grid cell in sequence, we would only need to reference the nearby grid quantities once for each grid cell. This means that in a code that has an average of ten particles per grid cell that the referencing of the memory could be an order of magnitude faster if the particles were grouped by grid cell. It is this reduction of memory "traffic" that was the primary motivation to develop a code that uses and maintains spatially ordered particles.

2.4 Parallelizing the Deposition

There is also a serious problem when we consider how a particle code could be parallelized. The *field interpolation*, the *push*, and the *solve* are relatively easy to parallelize because there is no data conflict. Unfortunately, there is a conflict in the *deposition interpolation* that must be resolved inorder to parallelize that phase of the calculation. The problem is that two different processors may try to update the same grid point at the same time. Until recently this many-to-one map problem also prevented vectorization. However, a new compiler for the C-90 now generates serial code that allows the computer to detect and correct any "collisions" at run time.

One approach to this problem claims that the errors generated by forcing the *deposition interpolation* to parallelize are relatively small and therefore acceptable. Such an "asynchronous" version would generate irreproducible results which is less than satisfactory even if the errors are small. We mention this possibility only because others have actually proposed it!

2.5 Parallel Sorting and Merging

We noted above that Horowitz [8,9] employed a special particle sorting scheme to allow parallel *deposition interpolation* within a sub-group of particles which the sorter selects such that no two members of the sub-group occupy the same cell. This technique removed the many-to-one mapping defect to allow concurrent deposition.

As suggested above, we want our particles ordered spatially and some kind of sorting and reordering of particle quantities will be required to maintain their order as the particles are moved in the simulation. And although it was efficient use of the memory that first lead us to sorting, it was the

realization that ordered particles would allow both domain decomposition and better vectorization that really convinced us to explore this avenue. Domain decomposition, as a well known paradigm for parallelizing algorithms, suggested how we could design our code to operate well in a parallel environment. Since these approaches were also suggested by our efforts to optimize our serial codes, we think that this avenue can lead to a "win-win" situation in which the most efficient serial code scales linearly to a most efficient parallel code. Of course, this whole argument fails if the sorting and reordering takes too much time.

Spatially ordered particles can be partitioned into local compact subgroups including all the particles within a given sub-domain. In such an arrangement, the *deposition interpolation* can be done concurrently by all the sub-domains because the many-to-one maps are all removed. And it is clear that the *field interpolation* and *push* are trivially parallelized in such an arrangement since there never was a mapping problem in the first place. Later we will show how the *field solve* also parallelizes nicely under this paradigm.

Moreover, within a sub-domain, the innermost loop of the *deposition interpolation* can be vectorized in direct vector mode. Likewise, the *field interpolation* and the *push* can have all the loops over the particles within a sub-domain processed in direct vector mode. This all leads us to believe that we can build a code that is both parallelized and vectorized which is likely to be optimal on a vector-parallel computer such as the Cray C-90.

Our approach to sorting the particles is based on the fact that only a relatively small fraction of the particles leave their former grid cells during any one push phase. We partition all moved particles into *transit* particles and *retained* particles according to whether they crossed into new grid cells or not.

We shall focus on the serial algorithm first and note its generalization to parallel computing later. Our sorting and rebuilding method is really a combination of a fast $n \log n$ sort on the n transit particles followed by a merge with the retained particles. The economy of our method is based on the fact that n is much smaller than the total number of particles N. Since the merge operation requires a number of operations proportional to N and since the work involved in the fast $n \log n$ sort is relatively small (in most cases) the resulting sort of the entire particle table requires work of order N operations. We believe that the cost of sorting would be prohibitive if we simply applied a standard sorting method to the entire particle table; it is the two step process of sorting the transit particles followed by the merge that gives us a combined method that is fast enough to be affordable.

An obvious problem with this method is the requirement that two particle tables must be maintained in memory. And, in fact, such a severe storage requirement would almost cause us to abandon this method except for the following consideration: There are other phases of the calculation that require large areas of scratch memory of roughly the same size as the particle table. As it will be shown later, we can effectively store these ar-

eas of scratch memory on top of whichever particle table is unused at any particular moment.

3 Vector-Serial HOPS Versions OSOP and RSOP

In this section we describe in more detail the algorithms and coding employed in the vector-serial version of HOPS. For all the details, the reader is referred to the source listing accompanying this paper or available from the program library.

3.1 Ordered Storage with Ordered Processing

The basic methodology used in HOPS, as previously stated, relies on the ordered storage of the particle information. We wish to distinguish this basic method by the acronym OSOP (Ordered Storage Ordered Processing) from a related method RSOP (Random Storage Ordered Processing), to be described later.

HOPS carries the particle variables ip, jp, and kp in the particle tables; these indices specify the coordinates of the grid cell containing the particle. It is convenient to also carry the offset index

$$l(n) = ip(n) + im * (jp(n) - 1) + im * jm * (kp(n) - 1) \qquad (1)$$

where im and jm are the grid dimensions in the x and y coordinates. The quantities ip, jp, kp and l are computed as the last part of the *push* phase.

3.1.1 Sorting and Reordering

The key methodologies to the success of the HOPS code are those of sorting, reordering, and rebuilding the particle tables. The cell index of particle n, $l(n)$, is the reference by which we define the spatial order of the particles. HOPS is designed to maintain $l(n)$ as a monotone non-decreasing sequence. After each *push* this ordering is broken; it is the job of the Subroutine Sormerge to put the particles back into order.

To prepare for the sorting and rebuilding, the last stage of the push subroutine scans the particle tables to find all particles that have crossed cell boundaries in the *push*. We do this by keeping the old value of l as l_{old} so we can test $(l - l_{old})$ to immeadiately determine which particles have been pushed out of their former cells (transit particles) and which ones have been retained in them. At this stage the transit particles are tested to see if they are escaping from the domain and then depending upon the boundary model imposed, they are either removed (lost) from the particle table or they are repositioned according to periodic or reflective boundary conditions. A two column transit table is constructed in which the first item gives $ictp(nt)$, the cell index of transit particle nt. In the second column is $iptn(nt) = n$ which is the pointer from transit particle nt to its position n

in the original particle table. As the particle table is scanned for the transit particles, we also indentify contiguous groups of *retained* particles by setting pointers to the bottom and top indices n of the retained groups. To do this we have another two column table with $isgbeg(nr)$ and $isgfin(nr)$ which give respectively the beginning and final index n of the nrth retained group; we call this the retained table. As we shall explain later, the preparatory work of determining the transit particles is done as part of the *push* because this is most expedient for parallel computing.

The routine Sormerge then proceeds to sort and rebuild the particle table as follows:

Since the *retained* particles did not change their cell indices, they remain properly ordered. The *transit* particles, on the other hand, are generally disordered. We apply the radix sorting method which is implemented in the CRI routine $ORDERS2$ to find the permutation $nti(nt)$ such that $ictp(nti(nt))$ is a monotone sequence in nt.

HOPS employs pointered arrays which enables dynamic management of the memory. We have allocated two blocks of memory for the storage of the particle tables. Rebuilding the particle table is done by moving (or reading) the data from the old "block" to the new one, as follows: Proceeding, a cell at a time, we start through the ordered *transit* table and move the *transit* particles' attributes from the old table followed by the *retained* particles. The result is a new particle table for which $l(n)$ is monotone. The present version of HOPS carries 17 attributes per particle. So when we read the data of a transit particle from the old block to the new one, it is a vector transfer of 17 words. And when we read the data of a contiguous group of *retained* particles we move $17 * (isgfin(nr) - isgbeg(nr) + 1)$ words of data in a vector transfer.

In a last pass over the particles we verify their correct ordering and we determine the pointer $nfpcl(iocl)$ which specifies the first particle in each occupied cell. Tests of this routine show that nearly all the work is spent in rebuilding the table. We have found it essential to move the retained particles in groups rather than individually because it approximately doubles the speed of this routine, Sormerge.

To recapitulate, as the last phase of the *push* we:

1. Construct a transit table of particles departing cells
2. Construct transit pointers to old particle table
3. Construct retained pointers to old particle table

Then Sormerge performs the following tasks:

1. Sorts the transit table ordering particles by cells
2. Allocates a new particle table
3. Processes the grid cells in ascending order and for each cell:

- Moves the particle data of transit particles to new table
- Moves the particle data of retained particles to new table

3.1.2 Ordered Field Interpolation and Push

In HOPS we combine these first two phases so that as soon as we determine the fields on the particles within a grid cell, we advance their velocities and positions. In order to properly time center the deposition phase of the previous time step, the particles were only advanced through half a time step before the deposition. Thus, before advancing the velocities in the new time step, the particle positions must be advanced through the second half of the time step as follows:

$$\mathbf{x}^n = \mathbf{x}^{n-1/2} + .5\mathbf{v}^{n-1/2}dt \tag{2}$$

Since this formula does not depend on any grid quantities, it leads to code which is trivially vectorized. Before proceeding to the velocity advance and the subsequent push, we must recompute the transit quantities and then reorder the particles by invoking the routine Sormerge, just described in Section **3.2.** We must do this because the halfstep of pushing can disorder the particles. Once, we have insured that we have ordered particles we proceed to advance the velocities. To do this we must determine the Lorentz accelerations (forces) on each particle; these are obtained by interpolation from the electric and magnetic fields on the grid. The determination of these forces and the subsequent push are accomplished in the routine Pushtran. Since the particles in each cell have been grouped together contiguously in the particle tables, we can proceed to process them cell by cell. Pushtran loops over all occupied grid cells. For each cell it obtains the force field components from the neighboring nearest eight vertices by four vector accesses of the interleaved fields. Each vector is of length twelve and holds the two 6-vectors of field values corresponding to two adjacent grid points in the x direction.

Within each cell we then loop over the contained particles and use the trilinear interpolation formula to establish the electromagnetic field components at each particle. The accelerations are determined and the velocities are advanced:

$$\mathbf{v}^{n+1/2} = \mathbf{v}^{n-1/2} + \mathbf{a}^n dt \tag{3}$$

Since $\mathbf{a}^n$ itself depends on $\mathbf{v}^{n+1/2}$ (via the Lorentz force) this leads to an implicit equation in $\mathbf{v}^{n+1/2}$, we apply the well known Boris push[11] which correctly advances the implicit form of Eq. 3 in a time and space centered manner.

In the second to last stage of Pushtran, we also perform the first half of the spatial advance. This is done in the same loop as the velocity update. It takes the form:

$$\mathbf{x}^{n+1/2} = \mathbf{x}^n + .5\mathbf{v}^{n+1/2}dt \tag{4}$$

Once this spatial advance is finished we must reorder the particle table. As described above, the last stage of Pushtran finds the transit particles and generates pointers relating them and the retained particle groups to the particle table. Then Sormerge is called again to sort and reorder the particles.

3.1.3 Ordered Deposition Interpolation

In the *deposition interpolation* a portion of each particle's charge and current contribution is assigned and accumulated at the neighboring grid points.

To get started we need to compute the grid locations just below the particle position; they are:

$\tilde{x}_n$ Lower grid point position in x

$\tilde{y}_n$ Lower grid point position in y

$\tilde{z}_n$ Lower grid point position in z

and from these we can construct the various interpolation weights. Then, for example,

$$W(n,p,1) = C_p(\tilde{x}_n + dx - xp_n)(\tilde{y}_n + dy - yp_n)(\tilde{z}_n + dz - zp_n) \tag{5}$$

gives the weight for the first vertex $(\tilde{x}_n, \tilde{y}_n, \tilde{z}_n)$ for the pth physical component as contributed by the nth particle. The last index of $W(n,p,v)$ is v which indicates which of the eight vertices is getting the deposit. The normalization coefficients C_p contain the charge and current factors together with appropriate geometric factors. C_1 gives the charge while C_2, C_3, C_4 give the x,y,z-components of the current contributions. Altogether, then, we compute 32 weights $W(n,p,v)$ for each n. The physical units of these weights are the actual charge density and current density contributions to be accumulated at the various grid points. These quantities are computed in direct vector mode in loops for which n is the loop index.

Instead of doing the deposition directly into the grid arrays for charge and current density, we alternatively use the eight partial grid arrays $Q(p,v,l(1,n))$ for $v = 1,8$. Since the leading index is the indicator of which physical quantity is being considered, the charge and current arrays are interleaved. Inorder to vectorize this accumulation we have an outer loop on the index v and an inner loop on p. We use $p = 1,8$ on the inner loop while v skips by two's through the outer loop; this means we combine the four contributions to two adjacent grid points into a vector loop length of eight. In Fortran the loop is:

```
do 40 iocl = 1, ioclmax
lfix = l(1,nfpcl(iocl))
```

```
      do 40 n = nfpcl(iocl), nfpcl(iocl+1)-1
cdir$ ivdep
      do 40 id = 1, 8
      Q(id,1,lfix+ioff(1)) = wt(n,id,1) + Q(id,1,lfix+ioff(1))
      Q(id,3,lfix+ioff(3)) = wt(n,id,3) + Q(id,3,lfix+ioff(3))
      Q(id,5,lfix+ioff(5)) = wt(n,id,5) + Q(id,5,lfix+ioff(5))
      Q(id,7,lfix+ioff(7)) = wt(n,id,7) + Q(id,7,lfix+ioff(7))
 40   continue
```

In coding this loop we had a choice of doing a gather-scatter (indirect) vectorization over the index n or in doing a directly indexed vector loop over id. Although the loop on n can be constructed to have a long vector length, we find it more efficient to do direct vectorization on the short vector. The index $lfix$ gives the cell address that is common for all particles referenced inside the last two loops; its use allows many fewer references to the memory locations of the array Q. The quantities $ioff(v)$ in the third index give the offsets that relate the grid index of a given vertex to the first vertex of any given cell. This loop is presented here in detail because it is the most time consuming loop in the entire HOPS code. The compiler directive `cdir$ ivdep` forces the CFT77 compiler on the C-90 to vectorize this loop.

Once all the Q values are determined, we simply add them to generate the charge and current densities on the grid. Thus for each grid point i, j, k we compute:

$$d(i,j,k) = d(i,j,k) + \sum_{v=1}^{8} Q(1,v,ig) \tag{6}$$

where $ig(i,j,k)$ is the expanded offset grid index. The three components of the current density cx, cy, and cz are determined in an analogous fashion.

3.2 Random Storage with Ordered Processing

The foregoing description of the OSOP version of HOPS as given in subsections 3.1, 3.2, 3.3, and 3.4 is closely related to the RSOP version which we now describe. We still use the concept of ordered particles and the ordering is precisely that of the OSOP version. The index n is still the ordered index (spatially), but it is not the index of the particle table. Instead, we use $n'(n)$ to give the particle table index for the ordered nth particle. As a result 1 is replaced with

$$l(n'(n)) = ip(n'(n)) + im * (jp(n'(n)) - 1) + im * jm * (kp(n'(n)) - 1) \tag{7}$$

The designation of RSOP (random storage ordered processing) relates directly to this form. The permutation mapping $n'(n)$ gives the storage locations of the ordered index n. Thus $l(n')$ is no longer a monotone sequence in n'; rather $l(n'(n))$ is monotone non-decreasing with respect to the ordered index n.

3.2.1 Sorting and Reordering

The RSOP analogue of the routine Sormerge is Psormerge. The "P" prefix indicates "P"ermutation mapping. Psormerge is a very similar routine to Sormerge. Instead of rebuilding the particle table, we rebuild the permutation table. The permutation table has, of course, only one attribute per particle while in comparison the particle table in use in the OSOP version of HOPS has 17 attributes per particle. In contrast to the steps of the routine Sormerge, given above, we have the following steps in Psormerge:

1. Construct a transit table of particles departing cells
2. Construct transit pointers to old permutation table
3. Construct retained pointers to old permutation table

Then Psormerge performs the following tasks:

1. Sorts the transit table ordering particles by cells
2. Allocates a new permutation table
3. Processes the grid cells in ascending order and for each cell:
   - Moves the permutation entries of transit particles to new table
   - Moves the permutation entries of retained particles to new table

Psormerge employs a significant number of indirectly indexed loops while Sormerge has mostly directly indexed loops which favors Sormerge with respect to the arithmetic speed. Psormerge, on the other hand, moves 1/17th as much data which one might think could be moved 17 times faster than in Sormerge. However, in Sormerge the data is moved as vectors of minimum length 17 in comparison to Psormerge where the vectors are of minimum length 1. In moving the data of a group of contiguous "retained" particles, say of length nrg, the longest vectors will be of length $17nrg$ in Sormerge, but only of length nrg in Psormerge. The vector efficiency of the longer vectors is so high that the move in Psormerge is only a few percent faster than that of Sormerge. Most of our testing has shown that Sormerge is generally about 20% faster than Psormerge.

3.2.2 Ordered Field Interpolation and Push

The ordered field interpolation and push proceeds as in the OSOP version with the important change that the index n is now replaced with the indirect index $n'(n)$. The particles are spatially ordered with respect to n but stored with respect to $n'(n)$. This means that each directly vectorized loop in the OSOP field interpolation and push is now replaced with an indirectly

vectorized loop. Thus the less efficient mechanisms of gather-scatter are used instead of those of direct vectorization.

From the point of view of efficient memory usage, both OSOP and RSOP make the same number of memory references but the former makes constant stride direct references while the latter makes random indirect references. The RSOP access not only requires the computation of the indirect index, but also is prone to memory bank conflicts. Althought there is some variability in the relative performance of the OSOP and RSOP *push* the OSOP tends to be about 10% faster on the Cray C-90 over a wide range of tests.

3.2.3 Ordered Deposition Interpolation

The *deposition interpolation* of the RSOP version proceeds very much like the OSOP version but it uses the indirect index $n'(n)$ instead of n.

Now the expression determining the weight factors is:

$$W(n,p,1) = C_p(\tilde{x}_n + dx - xp'_n(n))(\tilde{y}_n + dy - yp'_n(n))(\tilde{z}_n + dz - zp'_n(n)) \quad (8)$$

gives the weight for the first vertex $(\tilde{x}_n, \tilde{y}_n, \tilde{z}_n)$ for the pth physical component as contributed by the nth ordered particle which is particle $n'(n)$ in the particle table.

The main accumulation loop in the deposition, shown as Loop 40 above, remains the same as in the OSOP version. It has a direct vector structure even here in the RSOP version. This suggests that the RSOP deposition will perform comparably to the OSOP version. The RSOP deposition will be somewhat slower since the loops computing the weights $W(n,p,v)$ are now indrectly vectorized. But since the deposition loop dominates, the difference between RSOP deposition and that of OSOP will be minor. In typical test runs we have found the OSOP version about 3% faster, more or less confirming our estimates.

4 Vector-Parallel HOPS

Both versions of HOPS employ the same domain decomposition paradigm for deriving the parallel generalization of the basic HOPS algorithms described above. The computational domain is logically rectangular and three dimensional. Our domain decomposition uses rectangular subdomains constructed from the independent partitioning of the three coordinate axes. Most of the examples to be given use a domain constructed from four partitions in each coordinate which results in a set of 64 subdomains. HOPS parallelizes over these many subdomains.

In both versions the grid cells are ordered by a scheme we refer to as supra-natural ordering which means that the cells employ a natural ordering within each sub-domain while the global ordering among the sub-domains also has an ordering that is natural with respect to indices labelling the

subdomains. The important aspect of the grid cell ordering is that the cells within each subdomain form a contiguous subset of the total set of cells.

For the OSOP version, the particle table is also partitioned into contiguous subsets by virtue of the spatial ordering of the particles. In the RSOP version the particle table is not partitioned contiguously, but the permutation table (or pointer table) relating the ordered index to the serial index is partitioned contiguously.

In both versions, the grid data is contingously partitioned among the subdomains, but the possibility of conflicts at the interfaces between the subdomains requires some care.

4.1 Parallel Pushing

The pushing phase of the parallelized versions of HOPS is a very simple generalization of the serial version. It is only the properties of the particles that are modified together with the generation of the transit particles and their associated pointers. All these quantities are disjoint and there is no communication or information passed from the pushing phase of the other subdomains.

An outer loop is introduced which ranges over the particle subsets associated with each subdomain. The pointers *imsdbeg*(*isb*) and *imsdend*(*isb*) give the ordered particle index of the first and last particles in the *isb*th subdomain.

As in the serial versions, the last phases of the *push* construct transit particle quantities including pointers that give the particle table locations of each of them. While in the serial version, all the transit particles either reentered other cells or were lost at the boundaries, here there is a further possibility that some of the transit particles may enter other subdomains. To account for this we now introduce the class of emigrant particles which are identified and recorded just like the transit particles. The independence of each subdomain in the *push* prevents us from introducing the emigrants into their new subdomains during this phase. One of the emigrant quantities, the cell address of origin, is placed in a special two-dimensional array in which the row designates the the sub-domain and "serial" index of departure while the column designates the destination subdomain. Since each emigrant particle has only one destination, this table is largely filled with zeros, since each row can have only one entry. This rather sparse table- or augmented table- will be required for parallel operation in the yet to be described immigration process.

4.2 Parallel Sorting and Reordering

Once the augmented table of emigrant cell addresses is completed, it is possible to synchronize and perform all the sorting and reordering in parallel. Since we need only read (and not store into) this table, it is clear that no parallel conflicts are possible. And as each column contains the immigrant

quantities of a given desitination sub-domain, it is easy to scan down the columns, in parallel, to determine the inverse mappings (or pointers) which in turn allow us to "read" the immigrant quantities. We subsequently use them to extend the local subdomain transit tables. At this point each subdomain has a complete list of the particles now residing in its partition, comprised of retained groups and the transit particles.

The remainder of the parallel sort and reorder proceeds independently within the subdomains and consequently all of this work is performed in parallel. The actual algorithm is the same as in the serial version, only here it is applied to a subdomain.

4.3 Parallel Deposition

The parallel deposition relies on the spatial decomposition of domain. Grid points that lie on the interfaces between subdomains will get contributions to the grid quantities such as charge density from the adjoining subdomains; this means that attempts to update the grid quantities in parallel could result in conflicts. This is, of course, the many-to-one mapping problem we already encountered.

We handle this problem by exploiting the partial arrays *wfnew* which we already use in the serial versions. These arrays, it will be remembered, contain the eight fractional contributions of each particle to its eight enclosing grid cell vertices.

Since there are four physical quantities to be deposited and since there are eight nearby vertices to receive the contributions of each particle, the array *wfnew* should be dimensioned *32 nmax*. Conflicts cannot occur because the particle subsets contributing to anyone of these 32 arrays are disjoint. Ideally, one could build up the Q arrays totally in parallel and then once they are formed, synchronize and then sum the Q's in vector-parallel mode.

Unfortunately, storage limitations require that *wfnew* be smaller; it would be the largest array in the code otherwise and would be too large to store in the unused particle table. So we have limited its size to *8 nmax* which means we need to build up the Q's in four phases; each of the four phases proceeds in parallel and then a synchronization is needed to go on to the next phase. Each phase completes the contributions to two of the nearby vertices. The innermost loop in generating the Q's ranges over four quantities in direct vector mode. Although it is possible to re-arrange the loops such that the innermost loop admits indirect vectorization on a long vector, our measurments show that keeping the short direct vector loop innermost is fastest. The last phase of the deposition, the fifth, then sums the Q's to obtain the charge and current densities, again in vector-parallel mode.

5 The Vector-Parallel Field Solver

5.1 Pre-Maxwell Field Approximation

In this section we will give an overview of the system of equations solved by the HOPS code. In this hybrid code we use the particle-in-cell method, as given above, for the ions, and a zero temperature fluid model for the electrons. In effect, the electron momentum equation along with Maxwell's equations determine the electromagnetic fields. The model chosen for the electrons and the fields is motivated by the dominant physical effects in recent problems of interest. We could equally well do a full electromagnetic and relativistic treatment were that wanted. Our ability to parallelize the field solver would not be affected much by the choice of the model chosen for the electrons and fields.

The following set of equations will be the starting point of the discussion of the various approximations used in the equations solved by HOPS.

$$\nabla \times \mathbf{B} = \frac{4\pi}{c}\,\mathbf{J} + \frac{1}{c}\,\frac{\partial \mathbf{E}}{\partial t} \tag{9}$$

$$\nabla \times \mathbf{E} = -\frac{1}{c}\,\frac{\partial \mathbf{B}}{\partial t} \tag{10}$$

$$m_e\,\frac{d\,\mathbf{v_e}}{d\,t} = -e\left(\mathbf{E} + \frac{\mathbf{v_e} \times \mathbf{B}}{c}\right) \tag{11}$$

$$m_i\,\frac{d\,\mathbf{v_i}}{d\,t} = Z_i e\left(\mathbf{E} + \frac{\mathbf{v_i} \times \mathbf{B}}{c}\right) \tag{12}$$

Equations 9, 10, and 11 will be used to determine the electromagnetic fields. Equation 12 determines the dynamics of the ion macroparticles. The usual particle-in-cell method described above is used to advance the ion quantities. The electromagnetic field calculation is somewhat different from that used by Horowitz in QN3D.[8] It differs in that we include some finite electron mass effects, and secondly the numerical method used to solve the field equation is new.

There are two basic assumptions used in the derivation of the equations used here. They are; (1) Quasineutrality, and (2) the Darwin model (neglect of the transverse (solenoidal) part of the displacement current in Ampere's law). The motivation in using these assumptions is to eliminate fast time scale physics in both electrostatic plasma oscillations as well as purely electromagnetic modes.

Combining these assumptions is consistent with neglecting the entire displacement current in Ampere's equation. Thus this equation becomes

$$\nabla \times \mathbf{B} = \frac{4\pi}{c}\,\mathbf{J}\,. \tag{13}$$

Neglecting the longitudinal (irrotational) part of the displacment current eliminates the electron plasma oscillation, because the longitudinal part of $\mathbf{J_e}$ is forced to be equal to the longitudinal part of $\mathbf{J_i}$ which can only respond on the ion time scale. Thus the combination of quasineutrality together with the Darwin model leads to set set of equations which we refer to as pre-Maxwell, neglecting all displacement currents.

Some of the electron inertia effects can be included without too much difficulty. First an equation for $\mathbf{E}$ is obtained by deriving the moment equation for the time rate of change of the electron current. To do this we take the first moment of the Vlasov equation, which implicitly contains Eq. 11,

$$\frac{d\,\mathbf{J_e}}{d\,t} = \frac{\omega_{pe}^2}{4\pi}\mathbf{E} - \frac{e}{m_e c}\mathbf{J_e}\times\mathbf{B}\,. \tag{14}$$

Some terms have been neglected in this equation. They are terms that depend on $\mathbf{J}_e^2$, electron pressure gradient, and electron drag. Using the definition of the total current

$$\mathbf{J} = \mathbf{J}_i + \mathbf{J}_e \tag{15}$$

and Ampere's law in the Darwin limit, (Eq. 13) Eq. 14 can be re-written in terms of the time rate of change of the ion current,

$$\frac{d\left(\frac{c}{4\pi}\nabla\times\mathbf{B} - \mathbf{J}_i\right)}{d\,t} = \frac{\omega_{pe}^2}{4\pi}\mathbf{E} - \frac{e}{m_e c}\mathbf{J_e}\times\mathbf{B}\,. \tag{16}$$

The first term on the left hand side can be changed by commuting the time derivative of the curl of $\mathbf{B}$ into the curl of a time derivative of $\mathbf{B}$. This term then becomes a curl curl $\mathbf{E}$ term via Eq. 10.

An equation for the time derivative of the ion current can be obtained from the ion equation of motion Eq. 12

$$\frac{d\,\mathbf{J_i}}{d\,t} = \frac{{\omega_{pi}}^2}{4\pi}\mathbf{E} + \frac{Ze}{m_e c}\mathbf{J_i}\times\mathbf{B}\,. \tag{17}$$

As in the derivation of the electron current equation, Eq. 16, terms that depend on $\mathbf{J}_i^2$, ion pressure gradient, and ion drag have been neglected. Thus Eq. 16 becomes,

$$\nabla\times(\nabla\times\mathbf{E}) + \left(\frac{\omega_{pe}^2}{c^2} + \frac{\omega_{pi}^2}{c^2}\right)\mathbf{E} + \frac{4\pi e}{c^3}\left(\frac{1}{m_e} + \frac{Z}{m_i}\right)\mathbf{J_i}\times\mathbf{B} - \frac{e}{m_e c^2}(\nabla\times\mathbf{B})\times\mathbf{B} = 0\,. \tag{18}$$

This equation is evaluated at a time level between the n and $n+1$ level. The electric and magnetic field at this ˜ time level are given by,

$$\tilde{\mathbf{E}} = \alpha\mathbf{E}^{n+1} + (1-\alpha)\mathbf{E}^n \tag{19}$$

$$\tilde{\mathbf{B}} = \alpha\mathbf{B}^{n+1} + (1-\alpha)\mathbf{B}^n\,, \tag{20}$$

where the ~ indicates a "mix" of the $n+1$ and n time level determined by the parameter α. $\tilde{\mathbf{B}}$ terms that appear in this equation are replaced by $\tilde{\mathbf{E}}$ terms by using Eq. 13,

$$\frac{\mathbf{B}^{n+1}-\mathbf{B}^{n}}{\Delta t} = -c\nabla\times\tilde{\mathbf{E}}\,. \tag{21}$$

Using equations 19, 20 and 21 we obtain,

$$\tilde{\mathbf{B}} = \mathbf{B}^{n} - \alpha c\Delta t\nabla\times\tilde{\mathbf{E}}\,. \tag{22}$$

Combining these equations the equation for $\tilde{\mathbf{E}}$ becomes,

$$\nabla\times(\nabla\times\tilde{\mathbf{E}}) + \mathcal{A}\tilde{\mathbf{E}} + (\nabla\times\tilde{\mathbf{E}})\times\mathbf{I} + \mathcal{G}\nabla\times(\nabla\times\tilde{\mathbf{E}})\times\mathbf{B}^{n} = \mathbf{Q} \tag{23}$$

where,

$$\mathcal{A} = \frac{4\pi n_i^{n+\frac{1}{2}}e^2}{c^2}\left(\frac{Z}{m_i}+\frac{1}{m_e}\right) \tag{24}$$

$$\mathcal{G} = \frac{\alpha e\Delta t}{m_e c} \tag{25}$$

$$\mathbf{I} = \alpha c\Delta t\left\{\frac{4\pi e}{c^3}\left(\frac{Z}{m_i}+\frac{1}{m_e}\right)\mathbf{J}_i^{n+\frac{1}{2}} - \frac{e}{m_e c^2}\nabla\times\mathbf{B}^{n}\right\} \tag{26}$$

$$\mathbf{Q} = -\frac{4\pi}{c^3}\left(\frac{Z}{m_i}+\frac{1}{m_e}\right)\mathbf{J}_i^{n+\frac{1}{2}}\times\mathbf{B}^{n} + \frac{e}{m_e c^2}(\nabla\times\mathbf{B}^{n})\times\mathbf{B}^{n}\,. \tag{27}$$

5.2 Discrete System of Field Equations

We have used the Mathematica program to simplify the conversion of this equation into Fortran. This is done by writing out a Taylor series expansion of $\tilde{\mathbf{E}}$, $\mathbf{B}^{n}$ and $\mathbf{J}_i^{n+\frac{1}{2}}$. Using the Mathematica programs to do the curls and cross products we can represent Eq. 23 in terms of grid point quantities. In our case this involves 27 grid point quantities for each of the three vector components of this equation. The Mathematica program then extracts the coefficients which multiply the three components of $\tilde{\mathbf{E}}_{i,j,k}$. Thus the equation is given in matrix form as,

$$\mathbf{M}\cdot\tilde{\mathbf{E}}_{i,j,k} = \mathcal{R}(\tilde{\mathbf{E}}_{i\pm1,j\pm1,k\pm1})\,. \tag{28}$$

The matrix M is three by three multiplying the three components of $\tilde{\mathbf{E}}_{i,j,k}$. All of the other 26 grid point quantities for each component are on the right hand side of Eq. 28. This equation can be inverted to give

$$\tilde{\mathbf{E}}_{i,j,k}^{m+1} = \mathbf{M}^{-1}\cdot\mathcal{R}(\tilde{\mathbf{E}}_{i\pm1,j\pm1,k\pm1}^{m})\,. \tag{29}$$

This is the main equation used in the field solve iteration scheme. In each iteration of this equation the $\tilde{\mathbf{E}}$ on the right hand side is given. The iteration continues until some convergence criterion is met. When convergence is obtained, $\mathbf{E}^{n+1}$ is obtained from $\tilde{\mathbf{E}}$ using equation 19, and then the new $\mathbf{B}$ is determined by,

$$\mathbf{B}^{n+1} = \mathbf{B}^n - \frac{c\Delta t}{2} \nabla \times \left(\mathbf{E}^{n+1} + \mathbf{E}^n\right). \tag{30}$$

In all of the simulations presented in this paper we use time centering, that is $\alpha = \frac{1}{2}$. This is the "best" value since the ion current and density used in Eq. 23 are determined at this time level.

5.3 Vector-Parallel Algorithm for Field Equations

To solve the pre-Maxwell field equations we simply iterate 29. From the form of its right hand side, it is clear that there is coupling at subdomain boundaries to grid points in adjacent sub-domains. Inorder to ensure that these values are up to date, we must put a synchronization barrier at the end of each iteration. That is, as soon as all the $\tilde{\mathbf{E}}^m$ are computed from the previous iteration level, we can begin to evaluate 29 in parallel over the several subdomains.

At the domain boundaries, 29 clearly requires guard cells extending one grid point beyond the physical domain boundaries. At each iteration the appropriate boundary conditions are used to determine the $\tilde{\mathbf{E}}$ values at these guard points.

Since we are working on a three-dimensional domain, 29 was originally evaluated in a triply nested do-loop. The compiler on the Cray C-90 generates code that runs the inner loop in vector mode and the outermost loop in parallel mode. Since the number of grid points in any one direction can be relatively small it is possible to have load balancing problems. For example, in many of our tests we have 32 grid cells in each coordinate. This leads to 33 grid points in each coordinate. The compiler generates 33 processes to be run on the 16 processors; this has a theoretical maximum overlap of 11. To overcome this kind of problem we could require that the number of grid points in the x-coordinate be a multiple of 16, or we could try to parallelize over the combined two outermost loops. The C-90 compiler is unable to parallelize over these combined loops. We have replaced the outer two loops with a single loop over a new index which has pointers to the index pairs representing the two-dimensional subspace of the oringinal outermost loops. This structure parallelizes with the C-90 autotasking compiler. In the example given, we now have 33^2 processes on 16 processors with a theoretical overlap of 15.78, which we consider acceptable.

6 Test Problems

Our test problems fall into two categories. The first group, relevant to the development of the HOPS code, are used to show the relative performance of HOPS against more typical PIC codes and also to compare the OSOP version relative to the RSOP version. In these performance tests we use the parallel versions of OSOP and RSOP but run them in a single processor to get a good comparison with the PIC code which can only run on a single processor. The second group of test problems were run on all 16 processors in parallel mode to make an assessment of HOPS parallel capabilities and its scalability; in this group we compare OSOP versus RSOP.

6.1 Single Processor Performance Tests

The first pair of test problems is used to make comparisons between three code versions. A special PIC version was generated by taking the RSOP version and removing all of the code required for the sorting, reordering, and other procedures specific to ordered processing; we sometimes refer to this as the denatured version of HOPS. The first problem simulates the expansion of a plasma from a very small sphere centered in the middle of the domain; the second problem simulates a thermal constant density plasma. In the first one most of the grid cells are empty while in the second one nearly all the cells have particles in them. In the first problem the vectors are quite long as there are many particles in the grid cells that are occupied; the the second problem the vectors are relatively short because nearly every cell has some particles in it. In the following table we show the timings in terms of microseconds per particle per time step for the first (point source) problem:

HOPS C-90 Performance Unitasking μsec Point Source Problem (320k particles)				
Procedure	OSOP Parallel Version (64 Subdomains)	RSOP Parallel Version (64 Subdomains)	OSOP Parallel Version (1 Subdomain)	PIC Traditional Code (1 Subdomain)
Push	.38	.41	.38	1.9
Sorder	1.0	1.4	.85	1.4[1]
Deposition	.44	.50	.44	.49
Total*	1.9	2.3	1.7	3.8

* Excludes field solve timings.

Here the parallel OSOP version of HOPS is 26% faster than either the traditional PIC code or the parallel RSOP version. [1] Boundary particle treatment timing for the PIC version. In the HOPS versions the boundary treatment is part of the Sorder routine.

Now in the second problem, where we represent a uniform thermal plasma,

the particles are equally divided among the subdomains. In the first simulation we have 400k particles implying a vector length of about 12 for critical loops. In the following table it is the relatively slower performance in the push phase that keeps HOPS from running faster than the PIC code.

HOPS C-90 Performance Unitasking μsec Thermal Plasma Bath (400k particles)			
Procedure	OSOP Parallel Version (64 Subdomains)	OSOP Parallel Version (1 Subdomain)	PIC Traditional Code (1 Subdomain)
Push	1.6	1.6	1.1
Sorder	2.2	1.5	1.7[1]
Deposition	2.2	2.2	2.3
Total*	5.9	5.3	5.0

* Excludes field solve timings.
[1] Boundary particle treatment timing for the PIC version. In the HOPS versions the boundary treatment is part of the Sorder routine.

If we re-do the second problem with 1000k particles, with about 33 elements in the critical vectors, then the superior vector performance in HOPS allows it to run faster than the PIC version as is evident in the next table.

HOPS C-90 Performance Unitasking μsec Thermal Plasma Bath (1000k particles)			
Procedure	OSOP Parallel Version (64 Subdomains)	OSOP Parallel Version (1 Subdomain)	PIC Traditional Code (1 Subdomain)
Push	.75	.76	1.3
Sorder	1.4	1.1	1.7[1]
Deposition	1.1	1.1	1.2
Total*	3.2	3.0	4.1

* Excludes field solve timings.
[1] Boundary particle treatment timing for the PIC version. In the HOPS versions the boundary treatment is part of the Sorder routine.

Of particular curiosity, in the three preceding tables, is the time taken to treat the boundary particles in the PIC code. It is rather surprising that it takes a comparable amount of time to the entire sort and rebuilding process in the HOPS versions. This is partially explained by noting that in HOPS only the transit particles need to be tested and treated, while in comparison the standard PIC codce requires the testing of every particle to see if it is a boundary particle.

6.2 Parallel Execution Tests

Both versions of HOPS have been run in parallel on 16 processors on the Cray C-90. The runs we did above, simulating the thermal plasma, were done again with first 400k particles and then 1000k particles. The processor times are shown in the next two tables:

HOPS C-90 Parallel Performance μsec Thermal Plasma Bath (400k particles)		
Procedure	OSOP Parallel Version (64 Subdomains)	RSOP Parallel Version (64 Subdomains)
Push	2.0	2.7
Sorder	1.9	2.3
Deposition	2.8	2.7
Total*	6.7	7.7

HOPS C-90 Parallel Performance μsec Thermal Plasma Bath (1000k particles)		
Procedure	OSOP Parallel Version (64 Subdomains)	RSOP Parallel Version (64 Subdomains)
Push	1.1	1.8
Sorder	1.1	1.4
Deposition	1.5	1.5
Total*	3.7	4.7

Although the OSOP version has always been superior to the RSOP version on the Cray multiprocessor vector machines, we do not know if this will remain true on various massively parallel computers.

7 Conclusions

We have developed a new particle simulation code, HOPS, which has been tested on the Cray C-90. HOPS does not change the basic particle pushing algorithms but changes the way the "bookkeeping" is done. HOPS has been designed to remedy the well known inefficiencies of traditional particle codes, particularly the excessive accessing of memory, the degraded performance of gather-scatter vectorization, and the intractability of parallelization schemes. While other parallel particle codes[?] address some of these issues, only HOPS remedies all three.

While some parallel schemes coarse sort the particles by sub-domains, we fine sort them by grid cells. This allows us to treat all the particles in

a grid cell together which allows direct vector implementation while at the same time reducing the number of fetches of field quantities. We think that HOPS will also perform well on massively parallel computers- even those without vector capabilities. We believe that super-scalar processor chips will perform well on our directly indexed inner loops and further more we believe that cache hits will be maximized as a result of the particles being ordered within a subdomain (processor.)

As described above, we have shown HOPS to have a significant performance edge over traditional PIC simulation codes for those cases where we use small enough time steps to guarantee reasonable accuracy of the particle trajectories. It is only in cases where the time steps are large, corresponding to inaccurate trajectories, that the PIC methods are faster. Thus for reasonable simulations, HOPS is a superior algorithm on a single processor and has the added advantage that it scales linearly to large number of parallel processors.

Lastly, to hedge our bets we have developed the sibling methods OSOP (Ordered Storage Ordered Processing) and RSOP (Random Storage Ordered Processing) together so that perhaps one of these methods will be optimal on massively parallel machines. It is our prejudice that OSOP will retain its edge over RSOP as we move to the MPP platforms.

8 Acknowledgements

We are indebted to the late Oscar Buneman for his suggestions and for his encouragement. Oscar was also implementing an ordered storage scheme for PIC codes at the time of his death. We dedicate this paper to his memory. We thank Bruce Cohen, Bruce Curtis, Alex Friedman, Dennis Hewett, Bruce Langdon, Alan Mankofsky, and Timothy Williams for valuable advice on the many questions that arose in these studies. This work was supported by the U.S. Department of Energy for the Lawrence Livermore National Laboratory under contract W-7405-ENG-48. Cray-1 is a registered trademark and Cray-2, Cray X-MP, Cray Y-MP, and Cray C-90 are trademarks of Cray Research, Inc.

References

[1] A. Bruce Langdon and D. C. Barnes, "Direct Implicit Plasma Simulation," in *Multiple Time Scales*, Eds. Brackbill and Cohen, Acadmeic Press (1985), 337

[2] R. J. Mason, *J. Comput. Phys.* 41 (1981), 233

[3] A. Friedman, A. B. Langdon and B. I. Cohen, *Comments Plasma Phys. and Controlled Fusion*, 6 (1981), 225

[4] A. Bruce Langdon and Barbara F. Lasinski, "Electromagnetic and Relativistic Plasma Simulation Models," in *Methods in Computational Physics*, Ed. John Killeen, Academic Press (1976), 357

[5] J. P. Boris and D. L. Book, *J. Comput. Phys.* **11** (1973), 38

[6] S. ?. Zalesak, *J. Comput. Phys.* **31** (1979), 335

[7] O. Buneman, C.W. Barnes, J.C. Green, D.E. Nielsen, *J. Comput. Phys.* , **38** (1980), 1-44

[8] Eric J. Horowitz, Dan E. Shumaker and David V. Anderson, *J. Comput. Phys.* , **84** (1989), 279-310

[9] E. J. Horowitz, *J. Comput. Phys.* , **68** (1987), 56

[10] A. Heron and J. C. Adam, *J. Comput. Phys.* , **85** (1989), 284-301

[11] J. P. Boris, *Proc. 4th Conf. on the Numerical Simulation of Plasmas*, Naval Research Laboratory, Washington D.C. (1990), 3-67